# Graphics

Neha Patil

December 2020

## 1 Model

I have created a model of a hurricane. Each particle is a quad and has a texture (making each particle look like a sheep). All the rendering uses GPU rendering techniques.

I have implemented the ability to change characteristics of the systems while it is running, using key presses.

Things the user is able to control includes:

- Colour of the particles
- Life-time of the particles
- Wind

Run on MacBook Pro 2018 13 Inches.

## 2 Law of Motion

A tornado can be relatively realistically modelled as a 3-Dimensional vector field, where a particle's velocity is equal to a vector at time-step T and its position is its velocity * time-step.

The particle velocities to achieve an Archimedean spiral are as follows:

$$V_x = k \cdot \mathrm{e}^{n\theta} \cdot cos(2\theta)$$
$$V_y = k \cdot \mathrm{e}^{n\theta} \cdot sin(2\theta)$$
$$V_z = k \cdot \mathrm{e}^{n\theta}$$

Where k and n are constant factors which will be used to control the perceived wind in my simulation and $\theta$ is the starting angle of each particle.

The more particles there are the worse the approximation becomes. This is due to the fact the frame rate will decrease and therefore all the particles will begin to move as a polygon as the angle change theta will be changed in larger intervals. Compared the the smooth curved movement of a given point in a tornado. As the number of particles is linked to the frame rate the fuller tornadoes will not have very smooth curves.

The limited range of the random function I used also impacts the fidelity of my simulation as it limits the number of possible spirals generates, leading my tornado to resemble a spiral more than the smooth walls of a tornado.

(Plot of a random particles position when simulation is run with a high frame rate and a low frame rate shown below)

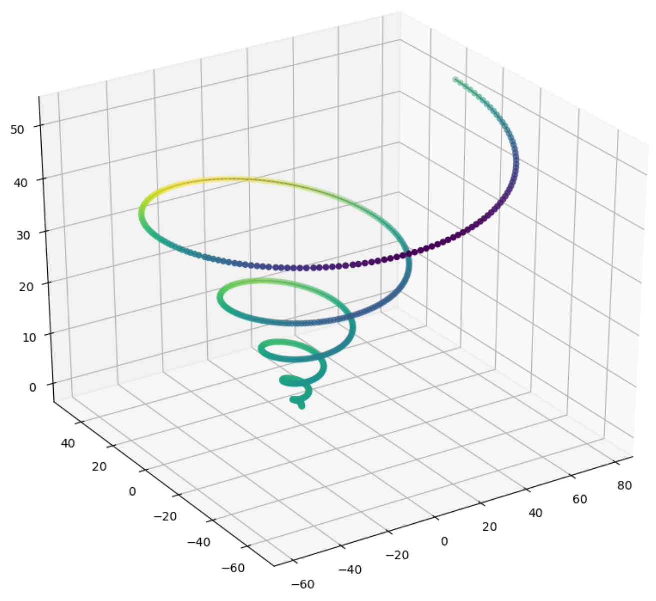

Figure 1: A plot of the position of a randomly chosen particle, with program running at a high frame rate.

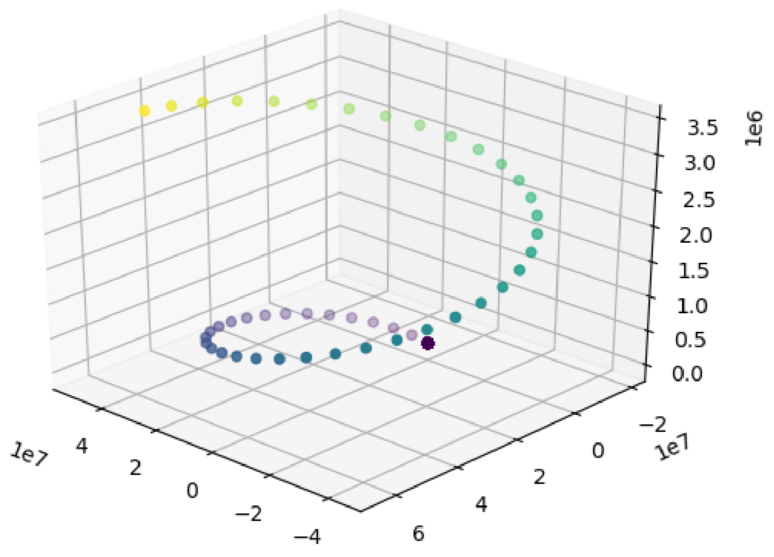

Figure 2: A plot of the position of a randomly chosen particle, with program running at a low frame rate.

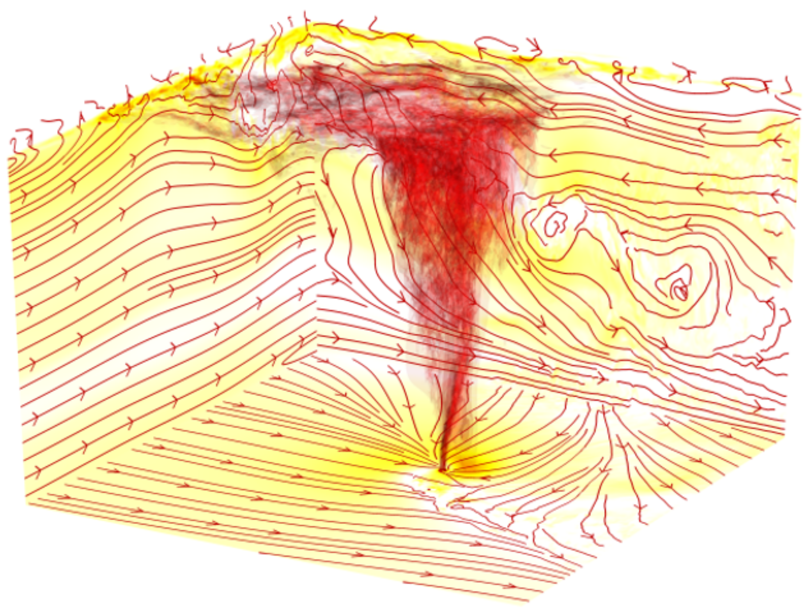

Figure 3: An example of a real Tornadoes vector field [1]

## 3 Efficiency of your approach

Instead of modelling a vector field, which would be inefficient, I have modelled the outside of the tornado by approximating the path of a single particle in the vector field and applying a general equation of a 3-D Archimedean spiral.

As I use floats for precision within my system, creating a look up table for a vector field would have created a very large file. I made my program more efficient by approximating the vector field using polar coordinate equations (as seen above). I randomised the starting theta of each particle as well as starting velocity allowing me to overlay different variations of Archimedean spiral to give the effect of depth and the characteristics of the vector field. Thus giving the illusion of a tornado in the most efficient way.

Another way I improved efficiency is by not updating particles which are dead. As I am modelling after a vector field, I differentiated the equations as acceleration. By changing the velocities of the particles rather than directly assigning the positions I was able to make my model truer to the vector field I was modelling. Therefore my implementation of a my law of motion is far more efficient than the original approach used to model tornadoes and is as close as possible to the motion I was modelling.

## 4 Analysis of overall performance

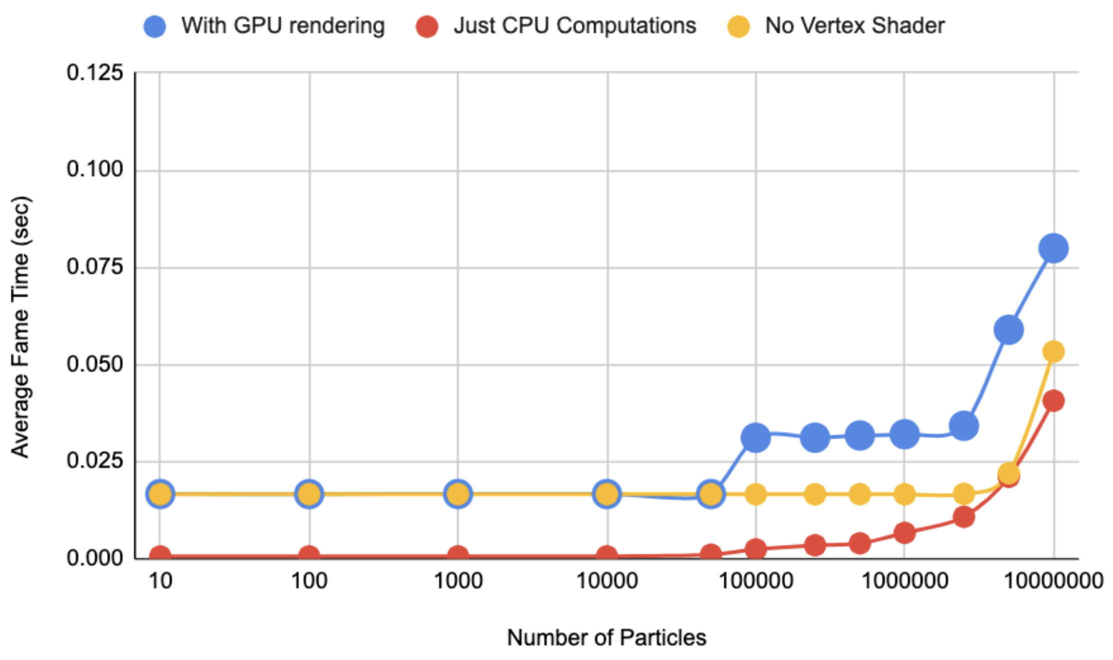

Figure 4: Experimental Graph

I believe my system is relatively efficient. I use a queue which is a very efficient data structure for my use as when I initialise my particles they are added to the queue in order and when I need to update particles which have died I am able to simply pop off the front of the queue in O(1), doing this allows me to avoid large amounts of particle processing, such as sorting them to find which particles have died which would at best be O(Nlog(N)). I am also able to exploit the ordering of my queue by using break statements to cut down computation e.g. To tell me when all the particles are taken up and therefore I can stop generating particles.

Another optimisation I have implemented is recycling particles by reassigning their attributes, rather than freeing dead particles and assigning memory for new ones.

As well as this I have implemented GPU-based approaches for particle rendering allowing my system to harness parallelized architecture. This addition allows my CPU to perform computations while GPU is rendering. I also use GLstream, the benefit being it is specialised for using data likely to be changed, compared to GL_static_draw.

A possible improvement to my code, if i were to using far more particles than my laptop is currently capable of handling, would be to keep an index of the number of particles currently being used and jump to that index when needing to fill my GPU buffers, reducing the number of iterations needed to do this.

My number of particles is limited my particle generation rate * lifetime of the particles therefore dynamically allocating memory (e.g. arrayList) would allow for smoother frame rate as the computation is much more manageable and less memory would be needed.

From the experimental data I have collated (shown above) I have discovered a few limitations to the performance of my program.

Firstly I found the frame rate is lower bounded to 0.0167s (3 s.f) even when only 1 particle is used, this is due to the refresh rate of my monitor being 60Hz, the monitor can show 60 different images per second and not more, therefore the frame time can not getting any lower than 1/60s = 0.0167s(3 s.f).

The graph I was able to produce also shows at low particle counts my program is bottle necked by the GPU. This is due to its low power as an integrated GPU. ( Shown by different rates of increase in performance time between simulation with and without rendering. ) As well as this at higher numbers of particles my GPU begins to make more of a performance impact ( shown by the jump in frame time of simulation with rendering at around 100,000 particles)

Hypothetically, I would be able to improve the performance of my simulation by getting a discrete GPU rather than using the less powerful integrated one. I would also need a monitor with higher refresh rate in order to get a better lower bound for frame time and for the particles to have smoother movement. Using a more powerful machine would also allow me to benefit from implementing multi-threading making my program more responsive key inputs as well as reduce system resource usage.

## Bibliography

[1] Vaclav Skala Michal Smolik and Zuzana Majdisova. *3D Vector Field Approximation and Critical Points Reduction Using Radial Basis Functions.* 2019 (cit. on p. 3).